Proposal for a Unified "Flux" N-tuple Format.

Robert Hatcher
FNAL/CD

October 2, 2012

Contents

1 Statement of Purpose

The FNAL neutrino experiments (MINOS, MINERνA, NOνA, ArgoNeut, MicroBooNe, LBNE) all have similar needs for simulations of the beamlines. Each of the NuMI, Booster and LBNE beamlines send protons into their respective targets, producing secondaries that decay to neutrinos; by keeping sufficient information those decays can be re-evaluated for different detector locations by event generators such as GENIE.

Various groups have used different tools to model the physics and geometry of the beamlines. These include combinations of GEANT3, GEANT4 and FLUKA. Unfortunately, over time, these simulations have come to have incompatible variants in the structure of their outputs. Some of these differences include a change of basic types, capitalization of the leaf element names, changes in array sizes, and additions of variables. This makes it more difficult for the different groups to make comparisons and to use common tools. GENIE's flux interface `GNuMIFlux` must support all the variants. This gets more difficult as individual, incompatible twists are introduced.

I am proposing that a single new format be defined and that all beamline simulations be modified to fill that format. The new structure should be an intelligent union of all the core parts and individual extensions. If a particular simulation doesn't generate or wish to store a non-essential element then they would flag it as unfilled. Additionally provisions would be made to use C++ STL `vector`s rather than fixed array sizes to allow for more flexibility and less waste. A scheme for proprietary (temporary) extensions should also be designed in to allow open-ended studies without the need for significant code changes. Below, I attempt to identify existing `Branch`es in the various `TTree`s and show their existing status and the new proposal.

It would also be useful to introduce a mechanism to record in the file some metadata that applies to the file as a whole. This includes total protons-on-target (rather than trying to infer it from the range of `evtno`); the actual detector locations used for "near" and "far"; and statements about the tools used to generate the file (e.g. flugg, geant4, etc. and build version).

The use of this ntuple format would be accompanied by the creation a new GENIE class `GDk2NuFlux`, replacing `GNuMIFlux` in normal use. The beamline agnostic name would avoid prejudice against Booster and LBNE beam simulations.

Thanks to Alex Himmel for producing MINOS-DocDB-6316 from whence I stole a lot of tables to serve as a starting point for this document.

2 Primary Ntuple

2.1 general characterisics

The primary ntuple holds entries representing decays that produced neutrinos with one entry for every neutrino recorded (generally with some importance weight). It is possible for the same initial proton to produce more than one entry (i.e. the same `evtno` might appear more than once).

The MINERνA variant of the `g4numi` layout appears to only add new branch elements which are discussed in Table 7.

simulation	**base program(s)**	**tree name**	**capitalization**	**char limit**
gnumi	geant3	h10	first char, sometimes	8 char
flugg	fluka + geant4	h10	follows gnumi	8 char
g4numi	geant4	nudata	studly, e.g. `NdxdzNear`	none
lbne	geant4	nudata	follows g4numi	none
—	— all —	dk2nu dkmeta	all lower case	none

Table 1: General properties of the ntuples.

At this time the format of any given ntuple file must be guessed from a combination of the file and tree names. By choosing a new unique tree name (e.g. `dk2nu`) for the new `TTree` format it can be easily identified; alternative suggestions for this name are welcome. I propose that branch element names for the new format are entirely lower case for ease of rememberence and typing. Also no artificial name cutoffs should be imposed (i.e. `ndxdznear` rather than `NdxdzNea`). 2¡ Each sub-section below tabulates a number of branch elements, gives their type for each `TTree` variant and a general description. These are grouped only for convenience and it is the aggregate that makes up the `TTree` structure.

Notes:

1. $\hat{z}$ is beam direction, centerline axis
2. energy & momentum are in GeV [allow to flag for MeV with `flagbits`? ‡]
3. distances in cm [allow to flag flag for m or mm with `flagbits`? ‡]
4. PDG particle codes [was Geant3, flag old with `flagbits`? ‡]
5. branch types: I=integer; F=float; D=double; TS=`TString`; S=STL `string`
6. $[n]$ = fixed size array; <>= STL vector
7. if type is ? then either type conflict or unknown whether final ntuple needs this element
8. † required for POT calculation
9. § required for weighting (e.g. relocation calculation of "x-y weight")

2.2 general entry info

Table 2 details some basic elements. The `run` branch is repetitive within a file but useful to distinguishing entries when the `TTree`s are chained together. Prior to the addition of any metadata to the file, the range of `evtno` values was used make a calculated *guess* at the total protons-on-target (POTs) the file represents. Because not every proton generates an entry in the `TTree` and because for some formats in some cases the proton number was lost (e.g. muon decays in flugg) one can not simply use the difference in the first and last entries.

Variable		g3	flugg	g4	lbne	new	Description
`run`		I	I	I	I	-	Run number (arbitrary)
`job`		-	-	-	-	I	Job number (arbitrary), replaces "run" to avoid "run period" confusion
`evtno`	†	I	I	I	I	-	Event number (proton on target)
`potnum`	†	-	-	-	-	I	proton on target number

Table 2: General entry information.

2.3 fixed decays

Table 3 represents the results of decays where the neutrino ray direction is either chosen randomly or forced through a particular point. The random decay is just that: whatever GEANT4 (or whatever) generated. The other tuples are calculated by limiting the ray to going through a given point. This choice will affect the neutrino's energy and direction and will have an associated weight (probability).

For a "far" detector far enough away that subtends a small enough solid angle the choice of a single point is relatively insignificant as the beam is essentially a parallel plane wave front. But this is not true for any sizable "near" detector which will see a line source rather than a point source and thus is subject to variation in energy spectra and intensity throughout its volume. Thus the "near" values can not be used as-is in event generators such as GENIE if they are to represent a detailed simulation. They are adequate for some crude purposes to get a general feel for different locations.

It is proposed to condense this section down to simple vectors of `nupx`, `nupy`, `nupz`, `nuenergy`, `nuwgt` where element [0] would represent the random decay (`nuwgt=1`), and subsequent elements hold some mixture of various "near" and "far" locations. Currently files lack any meta-data that tells one what location a "near" or "far" entry represents. For instance `flugg` files might have MINOS or NOνA locations used depending on who generated the file; this has led to surprises for the unwary and additional headaches when trying to rectify the differences seen by people running essentially the same code. By putting the exact location used (and named tags) into the meta-data and allowing arbitrary numbers of locations this can be avoided.

Variable	g3	flugg	g4	lbne	new	Description
`Ndxdz` `Ndydz`	F	D	D	F	-	ν direction slopes for a random decay
`Npz`	F	D	D	F	-	ν momentum (GeV/c) along the z-axis (beam axis) for a random decay
`Nenergy`	F	D	D	F	-	ν energy (GeV) for a random decay
`NdxdzNear` `NdydzNear`	F	D	D[11]	F[5]	-	Direction slopes for a ν forced towards the center of the "near" detector(s)
`NenergyN`	F	D	D[11]	F[5]	-	Energy for a ν forced towards the center of the "near" detector(s)
`NWtNear`	F	D	D[11]	F[5]	-	Weight for a ν forced towards the center of the "near" detector(s)
`NdxdzFar` `NdxdzFar`	F	D	D[2]	F[3]	-	Direction slopes for a ν forced towards the center of the "far" detector(s)
`NenergyF`	F	D	D[2]	F[3]	-	ν energy (GeV) for a decay forced to the center of the "far" detector(s)
`NWtFar`	F	D	D[2]	F[3]	-	ν weight for a decay forced to the center of the far detector(s)
`nupx` `nupy` `nupz`	-	-	-	-	<D>	ν momentum components for locations
`nuenergy` `nuwgt`	-	-	-	-	<D>	ν energy and weight for locations

Table 3: Limited neutrino ray information.

2.4 decay data

Table 4 is (mostly) the core information about the neutrino and the decay that gave rise to it. From the information marked with a § one can calculate the energy and weight for the neutrino ray to go through any point (small angles assumed??).

Variable		g3	flugg	g4	lbne	new	Description
Norig		I	I	I	I	I	neutrino origin: `g4numi`: 1=particle from target (or baffle), 2=from scraping, 3=from μ decay (Not filled in `flugg`)
Ndecay	¶	I	I	I	I	I	Decay process that produced the ν, see Table 11
Ntype	§	I	I	I	I	I	ν flavor. ‡PDG(GEANT) codes: $\nu_\mu 14(56), \bar{\nu}_\mu\text{-}14(55), \nu_e 12(53), \bar{\nu}_e\text{-}12(52)$
Vx Vy Vz	§	F	D	D	F	D	ν production vertex (cm)
pdPx pdPy pdPz	§	F	D	D	F	D	Momentum (GeV/c) of the ν parent at the ν production vertex (parent decay point)
ppdxdz ppdydz	§	F	D	D	F	D	Direction of the ν parent at its production point (which may be in the target)
pppz	§	F	D	D	F	D	z momentum (GeV/c) of the ν parent at its production point
ppenergy	§	F	D	D	F	D	Energy (GeV) of the ν parent at its production point
ppmedium	¶	I	I	D	F	?	Code for the material the ν parent was produced in (see Table 11)
ptype	§	I	I	I	I	I	ν parent species (GEANT codes‡)
ptrkid		-	-	-	I	?	**need lbne description**
ppvx ppvy ppvz		F	D	D	F	D	Production vertex (cm) of the ν parent
muparpx muparpy muparpz	§	F	D	D	F	D	Momentum (GeV/c) of the ν grandparent at the grandparent decay point (muons) or grandparent production point (hadrons) (at the decay point in production files – see footnote on page **??**
mupare	§	F	D	D	F	D	Energy (GeV) of the ν grandparent, as above
Necm	§	F	D	D	F	D	ν energy (GeV) in the center-of-mass frame
Nimpwt	§	F	D	D	D	D	Importance weight of the ν

Table 4: The core information about the decays.

2.5 parent data

Entries marked with a ¶ represent info (beyond §) MINOS or NOνA might use to in reweighting.

The `beamHWidth` through `hornCurrent` (and `protonN`) elements (found in the G4NUMI and G4LBNE layouts immediately after `evtno`) are presented here, out-of-order, because they seem related to others in this section. Most of those seem to be metadata (can anyone confirm this?) that won't vary from entry to entry. The `flugg`-only entries in Table 6 are derived values.

Variable	g3	flugg	g4	lbne	new	Description
xpoint ypoint zpoint	F	D	D	F	?	(Not filled in `flugg`, others?)
tvx tvy tvz	F	D	D	F	D	Position (cm) of the ν ancestor as it exits target (possibly, but not necessarily, the direct ν parent)
tpx tpy ¶ tpz	F	D	D	F	D	Momentum (GeV/c) of the ancestor as it exits target
tptype ¶	I	I	I	I	I	Species of the ancestor exiting the target (GEANT codes‡)
tgen	I	I	I	I	I	ν parent generation in cascade. 1 = primary proton, 2 = particles produced by proton interaction, 3 = particles from 2's
tgptype	I	I	-	-	?	Species of the parent of the particle exiting the target (GEANT codes‡)
tgppx tqppy tqppz	F	D	-	-	?	Momentum (GeV/c) of the parent of the particle exiting the target at the parent production point (at the decay point in production files – see footnote on page **??**
tprivx tprivy tprivz	F	D	-	-	?	Primary particle interaction vertex (not used)
beamx beamy beamz	F	D	-	-	?	Primary proton origin (cm)
beampx beampy beampz	F	D	-	-	?	Primary proton momentum (GeV/c)
protonN	-	-	-	I	?	**need lbne description of difference w/** `evtno`
beamHWidth beamVWidth	-	-	D	F	?	**need g4numi description**
beamX beamY	-	-	D	F	?	**need g4numi description**
protonX protonY protonZ	-	-	D	F	?	**need g4numi description**
protonPx protonPy protonPz	-	-	D	F	?	**need g4numi description**
nuTarZ	-	-	D	F	?	**need g4numi description**
hornCurrent	-	-	D	F	?	**need g4numi description**

Table 5: Miscellaneous information, mostly do to with some ancestors.

Variable	g3	flugg	g4	lbne	new	Description
`Vr`	-	D	-	-	?	$\sqrt{(}\texttt{Vx}^2 + \texttt{Vy}^2)$
`pdP`	-	D	-	-	?	$\sqrt{(}\texttt{pdPt}^2 + \texttt{pdPz}^2)$
`pdPt`	-	D	-	-	?	$\sqrt{(}\texttt{pdPx}^2 + \texttt{pdPy}^2)$
`ppp`	-	D	-	-	?	$\sqrt{(}\texttt{pppt}^2 + \texttt{pppz}^2)$
`pppt`	-	D	-	-	?	$\sqrt{(}\texttt{ppdxdz}^2 + \texttt{ppdydz}^2) \times \texttt{pppz}$
`ppvr`	-	D	-	-	?	**filled with `tvr` calculation, should be:** $\sqrt{(}\texttt{ppvx}^2 + \texttt{ppvy}^2)$
`muparp`	-	D	-	-	?	$\sqrt{(}\texttt{muparpt}^2 + \texttt{muparpz}^2)$
`muparpt`	-	D	-	-	?	$\sqrt{(}\texttt{muparpx}^2 + \texttt{muparpy}^2)$
`tvr`	-	D	-	-	?	**never filled! looks like typo stores calculated value in `ppvr`, should be:** $\sqrt{(}\texttt{tvx}^2 + \texttt{tvy}^2)$
`tp`	-	D	-	-	?	$\sqrt{(}\texttt{tpt}^2 + \texttt{tpz}^2)$
`tpt`	-	D	-	-	?	$\sqrt{(}\texttt{tpx}^2 + \texttt{tpy}^2)$

Table 6: `flugg` helper variables.

2.6 ancestor data

Table 7 is primarily `g4numi` and MINERνA's additions. Leo/? should verify the descriptions. By using STL `vector`s rather than fixed sized arrays we can eliminate the need for `ntrajectory` and `overflow`. Most of these need tweaks to the name to identify them as being information about the intermediate particles. Questions

- what do trackId and parentId represent? (trackId[n-1] = parentId[n] but is this just geant4 stack #?)
- Isn't start*[n] = stop*[n-1] (empirically seems to be true) ?
- choice of `TString` vs. STL `string`? (are these actually filled?)
- is entry [0] the proton (empirically true)?
- is entry [ntrajectory-1] the neutrino (empirically true)?
- indications in code that some of these entries use mm and MeV as units, which is at odds with the units for other variables

It would be nice to make the names a bit clearer that the represent the history between the proton and the neutrino. Or the group of variables could get pushed into a sub-object with a name such as `ancestors`.

Variable	g4	mnv	new	Description
`ntrajectory`	-	`I`	-	Number of intermediate levels *minerva check*
`overflow`	-	`B`	-	Flag list as incomplete *minerva check*
`pdg`	-	`I[10]`	-	Intermediate's particle type **descriptive name?**
`trackId`	-	`I[10]`	<I> ? -	??? **descriptive name? necessary?**
`parentId`	-	`I[10]`	<I> ? -	??? **descriptive name? necessary?**
`startx` `starty` `startz`	-	`D[10]`	<D>	??? Origin of intermediate **descriptive name?** *minerva difference w/* `trk` *above*
`stopx` `stopy` `stopz`	-	`D[10]`	<D>	??? End of intermediate **descriptive name?** *minerva check*
`startpx` `startpy` `startpz`	-	`D[10]`	<D>	??? Momentum at origin of intermediate **descriptive name?** *minerva difference w/* `trk` *above*
`stoppx` `stoppy` `stoppz`	-	`D[10]`	<D>	??? Momentum at end of intermediate **descriptive name?** *minerva check*
`pprodpx` `pprodpy` `pprodpz`	-	`D[10]`	<D>	??? **descriptive name?** *minerva check*
`proc`	-	`TS[10]`	<s>	??? process (at start or stop) **descriptive name?**
`ivol`	-	`TS[10]`	<s>	??? initial volume **descriptive name?**
`fvol`	-	`TS[10]`	<s>	??? final volume **descriptive name?**

Table 7: Information about intermediates between the proton and the decaying particle.

2.7 volume trajectory data

This group of variables provides crude tracking visualization by recording points where particles crossed volume boundaries. It is not clear what triggers the recording of a point.

Variable	g4	mnv	new	Description
`trkx` `trky` `trkz`	`D[10]`	`D[10]`	-	??? Position as (what?) particle crosses volume boundary **descriptive name?** *minerva check*
`trkpx` `trkpy` `trkpz`	`D[10]`	`D[10]`	-	??? Momentum as (what?) particle crosses volume boundary **descriptive name?** *minerva check*

Table 8: Information about positions in volume crossings.

2.8 proposed primary ntuple additions and metadata

Table 9 suggests some possible additions to the `dk2nu` tree. By providing STL `vectors` of integers and doubles users can add data that they need, especially for temporary short term studies, without having to change the basic format – which would affect all other users. The mapping from index into the vector to *meaning* will necessarily be up to the user. For cases where every entry has the same fixed mapping we would provide name vectors in the metadata to record that ordering. If the sizes vary on an entry by entry basis then it is left to the user to keep it straight.

I am also proposing the addition of a `flagbits` branch. My initial thoughts on this were to allow single bits to signal information. Some bits would be reserved for fixed purposes and and the rest would be up for individual user designation. One idea here would be to reserve bits to flag choices for units (currently these are expected to be cm for length, GeV for energy & momentum, but the user might prefer meters or mm and MeV) and particle codes (currently expected to be GEANT3 with ν extensions, but it would be nice to uniformly use PDG codes by default). While these suggested bits would generally be of file-wide scope the additional cost of one integer per entry is minimal.

Variable	new	Description
`vint`	<I>	STL `vector` of integers, for users to fill as they please
`vdbl`	<D>	STL `vector` of doubles, for users to fill as they please
`flagbits` ‡	I	Flags to indicate units and particle numbering scheme; some bits reserved for user designation

Table 9: Proposed additions for the primary ntuple (i.e. one entry per decay).

For the file-level metadata the proposal is that the object class be `dkmeta`. One could simply put one such object into every generated file, but it might be better to make this a tree in parallel with `dk2nu` which might facilitate chaining multiple files together and/or the concatenation of files.

Variable	**new**	**Description**
`job`	I	Identifying job # (replaces "run" to avoid "run period" confusion).
`pots`	D	Corresponding protons-on-target for the ntuple.
`beamsim`	s	Name and version of program that generated file (e.g. "g4numi/*tag*").
`physics`	s	Physics generator (e.g. "fluka08" or "g4.9.4p01").
`physcuts`	s	Tracking cuts (e.g. "threshold=0.1GeV").
`tgtcfg`	s	Target configuration (e.g. "minos/epoch3/-10cm").
`horncfg`	s	Horn configuration (e.g. "FHC/185A/LE/h1xoff=1mm").
`dkvolcfg`	s	Decay volume configuration (e.g. "helium" or "vacuum").
`beam0x` `beam0y`	D	Beam center position at start.
`beam0z`	D	Beam start z position.
`beamhwidth` `beamvwidth`	D	Beam horizontal and vertical widths.
`beamdxdz` `beamdydz`	D	Beam centerline slopes.

`xloc`
`yloc`
`zloc`
	<D>	Position info for each of the locations
		(beam system coordinates and units)

`nameloc`	<s>	Name strings for each of the locations
`vintnames`	<s>	STL `vector` of `string`s to hold names for `vint` elements.
`vdblnames`	<s>	STL `vector` of `string`s to hold names for `vdbl` elements.

Table 10: Proposed metadata elements (i.e. one entry per generated file).

3 Proposal

3.1 dk2nu.h

```
/**
 * \class dk2nu
 * \file  dk2nu.h
 *
 * \brief A class that defines the "dk2nu" object used as the primary
 *        branch for a TTree for the output of neutrino flux simulations
 *        such as g4numi, g4numi_flugg, etc.
 *
 * \author (last to touch it) $Author: rhatcher $
 *
 * \version $Revision: 1.1 $
 *
 * \date $Date: 2012/04/02 21:19:46 $
 *
 * Contact: rhatcher@fnal.gov
 *
 * $Id: dk2nu.h,v 1.1 2012/04/02 21:19:46 rhatcher Exp $
 *
 * Notes tagged with "DK2NU" are questions that should be answered
 */

#ifndef DK2NU_H
#define DK2NU_H

#include "TROOT.h"
#include "TObject.h"

#include <vector>
#include <string>

class dk2nu
{
private:
  ClassDef(dk2nu,3) // KEEP THIS UP-TO-DATE!  increment for each change

public:
  /**
   *   Public methods for constructing/destruction and resetting the data
   */
  dk2nu();
  virtual ~dk2nu();
  void Clear(const std::string &opt = ""); ///< reset everything to undefined

  /**
   *  All the data members are public as this class is used as a
   *  generalized struct, with just the addition of the Clear() method.
   *  As they will be branches of a TTree no specialized naming
   *  indicators signifying that they are member data of a class
   *  will be used, nor will any fancy capitalization schemes.
   */

  /**
   *=====================================================================
   *  General Info
   */
   Int_t job;              ///< identifying job #
   Int_t potnum;           ///< proton # processed by simulation

  /**
   *=====================================================================
   *  Fixed Decays:
   *  A random ray plus those directed at specific points.
   */
   std::vector<Double_t> nupx;     ///< px for nu at location(s)
   std::vector<Double_t> nupy;     ///< py for nu at location(s)
   std::vector<Double_t> nupz;     ///< pz for nu at location(s)
   std::vector<Double_t> nuenergy; ///< E for nu at location(s)
   std::vector<Double_t> nuwgt;    ///< weight for nu at location(s)

  /**
   *=====================================================================
   *  Decay Data:
   *  Core information about the neutrino and the decay that gave rise to it.
   *  % = necessary for reweighting
   */
   Int_t    norig;         ///< not used?
   Int_t    ndecay;        ///< decay process (see dkproc_t)
   Int_t    ntype;         ///< % neutrino flavor (PDG? code)

   Double_t vx;            ///< % neutrino production vertex x
   Double_t vy;            ///< % neutrino production vertex y
   Double_t vz;            ///< % neutrino production vertex z
   Double_t pdpx;          ///< % px momentum of nu parent at (vx,vy,vz)
   Double_t pdpy;          ///< % py momentum of nu parent at (vx,vy,vz)
   Double_t pdpz;          ///< % pz momentum of nu parent at (vx,vy,vz)

   /**  these are used in muon decay case? */
   Double_t ppdxdz;        ///< % direction of nu parent at its production point
   Double_t ppdydz;        ///< % direction of nu parent at its production point
   Double_t pppz;          ///< % z momentum of nu parent at its production point
   Double_t ppenergy;      ///< % energy of nu parent at its production point

   Double_t ppmedium;      ///< material nu parent was produced in
   Int_t    ptype;         ///< % nu parent species (PDG? code)

   /** momentum and energy of nu grandparent at
       muons:    grandparent decay point
       hadrons:  grandparent production point
       Huh?  this needs better documentation
    */
   Double_t muparpx;       ///< %
   Double_t muparpy;       ///< %
   Double_t muparpz;       ///< %
   Double_t mupare;        ///< % energy of nu grandparent
```

```

    Double_t necm;         ///< % nu energy in center-of-mass frame
    Double_t nimpwt;       ///< % production vertex z of nu parent

   /**
    *=====================================================================
    *  (Grand)Parent Info:
    *
    */

    /**
     * DK2NU: are these needed for any/all cases?
     */
    Double_t ppvx;         ///< production vertex x of nu parent
    Double_t ppvy;         ///< production vertex y of nu parent
    Double_t ppvz;         ///< production vertex z of nu parent

    /**
     * DK2NU: do we need these?  these aren't filled by flugg, others?
     */
    Double_t xpoint;       ///< ?
    Double_t ypoint;       ///< ?
    Double_t zpoint;       ///< ?

    /**
     * these ancestors are possibly, but not necessarily, the direct nu parent
     * DK2NU: can these be removed in favor of cascade info below?
     */
    Double_t tvx;          ///< x position of nu ancestor as it exits target
    Double_t tvy;          ///< y position of nu ancestor as it exits target
    Double_t tvz;          ///< z position of nu ancestor as it exits target
    Double_t tpx;          ///< x momentum of nu ancestor as it exits target
    Double_t tpy;          ///< y momentum of nu ancestor as it exits target
    Double_t tpz;          ///< z momentum of nu ancestor as it exits target
    Int_t    tptype;       ///< species of ancestor exiting the target
    Int_t    tgen;         ///< nu parent generation in cascade:
                           ///<   1=primary proton
                           ///<   2=particles produced by proton interaction
                           ///<   etc
    /**
     * these are only in g3numi and flugg
     * DK2NU: can these be removed in favor of cascade info below?
     *        for now we'll leave them in place
     */
    Int_t    tgptype;      ///< species of parent of particle exiting the target (PDG code?)

    Double_t tgppx;        ///< x momentum of parent of particle exiting target at the parent prod
    Double_t tgppy;        ///< y momentum
    Double_t tgppz;        ///< z momentum
    Double_t tprivx;       ///< primary particle interaction vtx (not used?)
    Double_t tprivy;       ///< primary particle interaction vtx (not used?)
    Double_t tprivz;       ///< primary particle intereaction vtx (not used?)
    Double_t beamx;        ///< primary proton origin
    Double_t beamy;        ///< primary proton origin
    Double_t beamz;        ///< primary proton origin
    Double_t beampx;       ///< primary proton momentum
    Double_t beampy;       ///< primary proton momentum
    Double_t beampz;       ///< primary proton momentum

    /**
     * these are in the g4numi and minerva ntuples
     * DK2NU: but what do they mean and are the duplicative to
     *        the more complete progenitor info below?
     */
    std::vector<Double_t> trkx;
    std::vector<Double_t> trky;
    std::vector<Double_t> trkz;
    std::vector<Double_t> trkpx;
    std::vector<Double_t> trkpy;
    std::vector<Double_t> trkpz;

   /**
    *======================================================================
    *  Progenitor Info:
    *  Complete ancestral info from primary proton down to decaying particle
    *
    *  DK2NU: this is mainly (based on) the minerva extensions *except*
    *         some names are changed to avoid confusion and
    *         distances will be cm, energies in GeV (unless the whole
    *         record uniformly uses something else and is flagged as such)
    */
    std::vector<Int_t>    apdg;     ///< ancestor species
    std::vector<Int_t>    trackid;  ///< ??? particle trackId
    std::vector<Int_t>    parentid; ///< ??? parentId

    std::vector<Double_t> startx;   ///< particle x initial position
    std::vector<Double_t> starty;   ///< particle y initial position
    std::vector<Double_t> startz;   ///< particle z initial position
    std::vector<Double_t> stopx;    ///< particle x final position
    std::vector<Double_t> stopy;    ///< particle y final position
    std::vector<Double_t> stopz;    ///< particle z final position

    std::vector<Double_t> startpx;  ///< particle x initial momentum
    std::vector<Double_t> startpy;  ///< particle y initial momentum
    std::vector<Double_t> startpz;  ///< particle z initial momentum
    std::vector<Double_t> stoppx;   ///< particle x final momentum
    std::vector<Double_t> stoppy;   ///< particle y final momentum
    std::vector<Double_t> stoppz;   ///< particle z final momentum

    std::vector<Double_t> pprodpx;  ///< parent x momentum when producing this particle, MeV/c
    std::vector<Double_t> pprodpy;  ///< parent y momentum when producing this particle
    std::vector<Double_t> pprodpz;  ///< parent z momentum when producing this particle

    std::vector<std::string> proc;  ///< name of the process that creates this particle

    std::vector<std::string> ivol;  ///< name of the volume where the particle starts
    std::vector<std::string> fvol;  ///< name of the volume where the particle stops

```

```
  /**
   *============================================================================
   *  Special Info:
   */
  Int_t    flagbits;      ///< bits signify non-std setting such as
                          ///< Geant vs. PDG codes, mm vs. cm, Mev vs. GeV
  std::vector<Int_t>    vint;    ///< user defined vector of integers
  std::vector<Double_t> vdbl;    ///< user defined vector of doubles

  /**
   *============================================================================
   *  Random Info:
   *  blah, blah, blah
   */

  Int_t    ptrkid;        ///< lbne addition

  /**
   *============================================================================
   *  Specialized enumerations
   */

  /**
   *  Proposed flag bits:
   */
  typedef enum flgbitval {
    flg_dist_m           = 0x00000000,  ///< no special bit for meters
    flg_dist_cm          = 0x00020000,  ///< distances in cm (default)
    flg_dist_mm          = 0x00030000,  ///< distances in mm
    flg_e_gev            = 0x00000000,  ///< no special bit for GeV (default)
    flg_e_mev            = 0x00300000,  ///< energies in MeV
    flg_usr_mask         = 0x0000FFFF,
    flg_reserved_mask    = 0xFFFF0000
  } flgbitval_t;

  /**
   *  Enumeration of decay processes, stored in "ndecay"
   *  store as integer; these are for reference
   *  DK2NU:  should there be an associated AsString() method
   *          that returns a text (optionally formatted for latex?)?
   */
  typedef enum dkproc {
    dkp_unknown         =  0,
    dkp_k0l_nuepimep    =  1,  ///< k0long => nu_e + pi- + e+
    dkp_k0l_nuebpipem   =  2,  ///< k0long => nu_e_bar + p+ + e-
    dkp_k0l_numupimmup  =  3,  ///< k0long => nu_mu + pi- + mu+
    dkp_k0l_numubpipmum =  4,  ///< k0long => nu_mu_bar + pi+ + mu-
    dkp_kp_numumup      =  5,  ///< k+ => nu_mu + mu+
    dkp_kp_nuepi0ep     =  6,  ///< k+ => nu_e + pi0 + e+
    dkp_kp_numupi0mup   =  7,  ///< k+ => nu_mu + pi0 + mu+
    dkp_kp_numubmum     =  8,  ///< k- => nu_mu_bar + mu-
    dkp_kp_nuebpi0em    =  9,  ///< k- => nu_e_bar + pi0 + e-
    dkp_kp_numubpi0mum  = 10,  ///< k- => nu_mu_bar + pi0 + mu-
    dkp_mup_nusep       = 11,  ///< mu+ => nu_mu_bar + nu_e + e+
      dkp_mum_nusep       = 12,  ///< mu- => nu_mu + nu_e_bar + e-
      dk_pip_numumup      = 13,  ///< pi+ => nu_mu + mu+
      dk_pim_numubmum     = 14,  ///< pi- => nu_mu_bar + mu-
      dkp_maximum,               ///< one-beyond end for iterating
      dkp_other           = 999, ///< flag for unusual cases
    } dkproc_t;

 };

#endif
```

3.2 dkmeta.h

```
/**
 * \class dkmeta
 * \file  dkmeta.h
 *
 * \brief A class that defines the "dkmeta" object used as the
 *        branch for a TTree for the output of meta-data from
 *        neutrino flux simulations such as g4numi, g4numi_flugg, etc.
 *        This tree has one entry of this type for the file.  Kept
 *        as a tree so files can be chained.
 *
 * \author (last to touch it) $Author: rhatcher $
 *
 * \version $Revision: 1.1 $
 *
 * \date $Date: 2012/04/02 21:19:46 $
 *
 * Contact: rhatcher@fnal.gov
 *
 * $Id: dkmeta.h,v 1.1 2012/04/02 21:19:46 rhatcher Exp $
 *
 * Notes tagged with "DKMETA" are questions that should be answered
 */

#ifndef DKMETA_H
#define DKMETA_H

#include "TROOT.h"
#include "TObject.h"

#include <vector>
#include <string>

class dkmeta
{
private:
  ClassDef(dkmeta,3) // KEEP THIS UP-TO-DATE!  increment for each change

public:
  /**
   *   Public methods for constructing/destruction and resetting the data
   */
```

```
    dkmeta();
    virtual ~dkmeta();
    void Clear(const std::string &opt = ""); ///< reset everything to undefined

    /**
     *  All the data members are public as this class is used as a
     *  generalized struct, with just the addition of the Clear() method.
     *  As they will be branches of a TTree no specialized naming
     *  indicators signifying that they are member data of a class
     *  will be used, nor will any fancy capitalization schemes.
     */

    /**
     *=====================================================================
     *  General Info:
     */
     Int_t    job;            ///< identifying job # (keep files distinct)
     Double_t pots;           ///< protons-on-target

     /**
      * DKMETA:
      * formatted strings are most flexible ...
      * but not necessarily convenient to use
      * ??? Should parts of these be standardized ???
      */
     std::string beamsim;    ///< e.g. "flugg" or "g4numi/<tag>"
     std::string physics;    ///< e.g. "fluka08", "g4.9.3p01"
     std::string physcuts;   ///< tracking cuts    e.g. "threshold=0.1GeV"
     std::string tgtcfg;     ///< target config    e.g. "minos/epoch3/-10cm"
     std::string horncfg;    ///< horn config      e.g. "FHC/185A/LE/h1xoff=1mm"
     std::string dkvolcfg;   ///< decay vol config e.g. "helium" or "vacuum"

    /**
     *=====================================================================
     *  Beam Info:
     */
     Double_t beam0x;       ///< x of beam center at start
     Double_t beam0y;       ///< y of beam center at start
     Double_t beam0z;       ///< z of beam start
     Double_t beamhwidth;   ///< horizontal width of beam
     Double_t beamvwidth;   ///< vertical width of beam
     Double_t beamdxdz;     ///< beam slope dx/dz
     Double_t beamdydz;     ///< beam slope dy/dz

    /**
     *=====================================================================
     *  Detector Position Info:
     *  Values are in beam coordinate system w/ units of "cm"
     */
     std::vector<Double_t> xloc;   ///< x positions of detectors
     std::vector<Double_t> yloc;   ///< y positions of detectors
     std::vector<Double_t> zloc;   ///< z positions of detectors

     std::vector<std::string> nameloc; ///< names of detector locations (e.g. "NOvA-ND-3x3")
```

```

   /**
    *========================================================================
    *  Special Info:
    *  Document extensibility enhancements
    */
    std::vector<std::string> vintnames;    ///< names of elements for user defined vector of integ
    std::vector<std::string> vdblnames;    ///< names of elements for user defined vector of doubl

  };

#endif
```

4 Example test program for filling

This example demonstrates the basics of creating and filling the ntuple. It also demonstrates the use of standardized code for reading locations, and generating weights for those locations. Extending the basic tree (for special non-standard additions) can be done without modifying the basic class; this is demonstrated using the `nonstd` class.

4.1 test_fill_dk2nu.C

```
/// \file  test_fill_dk2nu.C
/// \brief Test creating and filling a TTree based on:
///     dk2nu.h  (dk2nu.C)  - decays of beam particles to a neutrino
///     dkmeta.h (dkmeta.C) - metadata for the file
///
/// also show the use of reading location information, generating
/// weights at those locations, and how to extend the tree (for one-off
/// tests) without modifying the standard class.
///
/// This script can be run using:
///       root -b -q test_fill_dk2nu.C+
///
/// \author  Robert Hatcher <rhatcher \at fnal.gov>
///          Fermi National Accelerator Laboratory
///
/// \created   2012-04-03
/// \modified  2012-10-03
/// \version $Id: test_fill_dk2nu.C,v 1.1 2012/10/03 23:17:59 rhatcher Exp $
///============================================================================

#include <iostream>
#include <iomanip>
#include "TFile.h"
#include "TTree.h"
#include "TRandom3.h"

#include "dk2nu.h"
#include "dkmeta.h"

/// include these because we're not linking to anything external
/// so we need to include the source for dk2nu::Clear() and dkmeta::Clear()
#include "dk2nu.cc"
#include "dkmeta.cc"

/// example class for extending the tree with non-standard extras
/// that doesn't require modifying the basic "dk2nu" class
class nonstd {
  public:
    nonstd() { }
    virtual ~nonstd() { }
    void Clear() { }
    double foo;  ///< data for my one-off test
    double bar;  ///< more data
  ClassDef(nonstd,1)
};
```

```

/// make a dictionary for classes used in the tree
/// again do this because we have no external linkages to libraries
#ifdef __CINT__
#pragma link C++ class dk2nu+;
#pragma link C++ class dkmeta+;
#pragma link C++ class nonstd+;
#endif

/// include standardized code for reading location text file
#include "readWeightLocations.C"

/// include standardized code for getting energy/weight vectors for locations
#include "calcLocationWeights.C"

// flugg 500K POT lowth files seem to have 510000 as an upper limit on
// # of entries.   So to test for estimate of file size one needs to have
// that many entries _and_ semi-sensible values for all branches (so
// compression isn't better than it would be in real life).
void test_fill_dk2nu(unsigned int nentries=1000)
{

  // stuff...
  TRandom3* rndm = new TRandom3();

  ///-----------------------------------------------------------------------
  ///
  ///  equivalent to NumiAnalysis::NumiAnalysis() in g4numi
  ///
  ///-----------------------------------------------------------------------

  // create objects
  dk2nu*  dk2nuObj  = new dk2nu;
  dkmeta* dkmetaObj = new dkmeta;
  nonstd* nonstdObj = new nonstd;

  // read the text file for locations, fill the dkmeta object
  std::string locfilename = "locfile.txt";
  readWeightLocations(locfilename,dkmetaObj);

  // print out what we have for locations
  size_t nloc = dkmetaObj->nameloc.size();
  std::cout << "Read " << nloc << " locations read from \""
            << locfilename << "\"" << std::endl;
  for (size_t iloc = 0; iloc < nloc; ++iloc ) {
    std::cout << "{" << std::setw(10) << dkmetaObj->xloc[iloc]
              << "," << std::setw(10) << dkmetaObj->yloc[iloc]
              << "," << std::setw(10) << dkmetaObj->zloc[iloc]
              << " } \"" << dkmetaObj->nameloc[iloc] << "\""
              << std::endl;
  }

  ///-----------------------------------------------------------------------
  ///
  ///  equivalent to NumiAnalysis::book() in g4numi
  ///
  ///----------------------------------------------------------------------

  // create file, book tree, set branch address to created object
  TFile* treeFile = new TFile("test_dk2nu.root","RECREATE");

  TTree* dk2nu_tree = new TTree("dk2nu","FNAL neutrino ntuple");
  dk2nu_tree->Branch("dk2nu","dk2nu",&dk2nuObj,32000,1);
  // extend the tree with additional branches without modifying std class
  dk2nu_tree->Branch("nonstd","nonstd",&nonstdObj,32000,1);

  TTree* dkmeta_tree  = new TTree("dkmeta","FNAL neutrino ntuple metadata");
  dkmeta_tree->Branch("dkmeta","dkmeta",&dkmetaObj,32000,1);

  int myjob = 42;  // unique identifying job # for this series

  ///----------------------------------------------------------------------
  ///
  ///  equivalent to NumiAnalysis::? in g4numi
  ///  this is the main loop, making entries as the come about
  ///
  ///----------------------------------------------------------------------
  // fill a few element of a few entries
  for (unsigned int ipot=1; ipot <= nentries; ++ipot) {

    ///
    ///  equivalent to NumiAnalysis::FillNeutrinoNtuple() in g4numi
    ///  (only the part within the loop over ipot)
    ///

    // clear the object in preparation for filling an entry
    dk2nuObj->Clear();

    // fill with info ... only a few elements, just for test purposes
    dk2nuObj->job    = myjob;
    dk2nuObj->potnum = ipot;

    // pick a bogus particle type to decay, and a neutrino flavor
    int ptype = 211;  // pi+
    if ( ipot %  5 == 0 ) ptype = 321;  // k+
    if ( ipot % 50 == 0 ) ptype = 13;   // mu-
    int ntype = ( ( ptype == 321 ) ? 12 : 14 );
    TVector3 p3nu(1,2,3); // bogus random neutrino decay vector

    // calcLocationWeights needs these filled if it isn't going assert()
    // really need to fill the other bits at this point as well:
    //   ntype, ptype, vx, vy, vz, pdpx, pdpy, pdpz, necm,
    //   ppenergy, ppdxdz, ppdydz, pppz,
    //   muparpx, muparpy, muparpz, mupare
    dk2nuObj->ptype  = ptype;
    dk2nuObj->ntype  = ntype;

    // fill nupx, nupy, nupz, nuenergy, nuwgt(=1) for random decay
```

```
      // should be the 0-th entry
      if ( dkmetaObj->nameloc[0] == "random decay" ) {
        dk2nuObj->nupx.push_back(p3nu.x());
        dk2nuObj->nupy.push_back(p3nu.y());
        dk2nuObj->nupz.push_back(p3nu.z());
        dk2nuObj->nuenergy.push_back(p3nu.Mag());
        dk2nuObj->nuwgt.push_back(1.0);
      }
      // fill location specific p3, energy and weights; locations in metadata
      calcLocationWeights(dkmetaObj,dk2nuObj);

      // test the filling of vector where entries vary in length
      // ... really need to fill whole dk2nu object
      unsigned int nancestors = rndm->Integer(12) + 1;  // at least one entry
      for (unsigned int janc = 0; janc < nancestors; ++janc ) {
        int xpdg = rndm->Integer(100);
        dk2nuObj->apdg.push_back(janc*10000+xpdg);
      }

      // push a couple of user defined values for each entry
      dk2nuObj->vint.push_back(42);
      dk2nuObj->vint.push_back(ipot);

      // fill non-standard extension to tree with user additions
      nonstdObj->foo = ptype + 1000000;
      nonstdObj->bar = ipot + ptype;

      // push entry out to tree
      dk2nu_tree->Fill();

    } // end of fill loop

    ///-------------------------------------------------------------------------
    ///
    ///  equivalent to NumiAnalysis::finish() in g4numi
    ///
    ///-------------------------------------------------------------------------

    /// fill the rest of the metadata (locations filled above)
    //no! would clear location info // dkmetaObj->Clear();
    dkmetaObj->job  = myjob;  // needs to match the value in each dk2nu entry
    dkmetaObj->pots = 50000;  // ntuple represents this many protons-on-target
    dkmetaObj->beamsim = "test_fill_dk2nu.C";
    dkmetaObj->physics = "bogus";
    dkmetaObj->vintnames.push_back("mytemp_42");
    dkmetaObj->vintnames.push_back("mytemp_ipot");
    // push entry out to meta-data tree
    dkmeta_tree->Fill();

    // finish and clean-up
    treeFile->cd();
    dk2nu_tree->Write();
    dkmeta_tree->Write();
    treeFile->Close();
```

```
  delete treeFile; treeFile=0;
  dk2nu_tree=0;
  dkmeta_tree=0;
}
```

4.2 readWeightLocations.C

Simulation code would no longer hardcode location information into the source; instead the desired positions would be read from a simple text file.

```
#include <string>
#include <iostream>
#include <fstream>
#include <iomanip>

#include "dkmeta.h"

/// Read a text file that contains a header line followed by
/// quartets of "<xpos> <ypos> <zpos> <text string>" on separate
/// lines.  Fill the supplied vectors.  Trim off leading/trailing
/// blanks and quotes (single/double) from the string.
/// Convention has it that positions are given in (cm).
void readWeightLocations(std::string locfilename,
                         std::vector<std::string>& nameloc,
                         std::vector<double>& xloc,
                         std::vector<double>& yloc,
                         std::vector<double>& zloc)
{

  std::ifstream locfile(locfilename.c_str());

  int iline=0;

  // read/skip header line in text file
  char header[1000];
  locfile.getline(header,sizeof(header));
  ++iline;

  // read lines
  char tmp[1001];
  size_t tmplen = sizeof(tmp);
  while ( ! locfile.eof() ) {
    double x, y, z;
    locfile >> x >> y >> z;
    locfile.getline(tmp,tmplen-1);
    size_t i = locfile.gcount();
    // make sure the c-string is null terminated
    size_t inull = i;
    //if ( inull < 0 ) inull = 0;
    if ( inull > tmplen-1 ) inull = tmplen-1;
    tmp[inull] = '\0';
    std::string name(tmp);
    // ignore leading & trailing blanks (and any single/double quotes)
    size_t ilast  = name.find_last_not_of(" \t\n'\"");
    name.erase(ilast+1,std::string::npos);  // trim tail
    size_t ifirst = name.find_first_not_of(" \t\n'\"");
    name.erase(0,ifirst);  // trim head

    ++iline;
    if ( ! locfile.good() ) {
      //if ( verbose)
      //  std::cout << "stopped reading on line " << iline << std::endl;
      break;
    }
    nameloc.push_back(name);
    xloc.push_back(x);
    yloc.push_back(y);
    zloc.push_back(z);
  }

}

/// a variant that will fill the dkmeta object
void readWeightLocations(std::string locfilename, dkmeta* dkmetaObj)
{
  ///  read & print the locations where weights are to be calculated
  std::vector<std::string>& nameloc = dkmetaObj->nameloc;
  std::vector<double>& xloc         = dkmetaObj->xloc;
  std::vector<double>& yloc         = dkmetaObj->yloc;
  std::vector<double>& zloc         = dkmetaObj->zloc;

  /// make an entry for the random decay
  nameloc.push_back("random decay");
  xloc.push_back(0);  // positions for random case are bogus
  yloc.push_back(0);
  zloc.push_back(0);

  /// read and parse the text file for additional positions
  /// use the vector version
  readWeightLocations(locfilename, nameloc, xloc, yloc, zloc);
}
```

4.3 calcLocationWeights.C

Standardized code for calculating weights for detector positions.

```
#include <iostream>
#include <cassert>

#include "dkmeta.h"
#include "dk2nu.h"

#include "TMath.h"
#include "TVector3.h"

// forward declaration
int CalcEnuWgt(const dk2nu* dk2nuObj, const TVector3& xyz,
               double& enu, double& wgt_xy);

// user interface
```

```
void calcLocationWeights(dkmeta* dkmetaObj, dk2nu* dk2nuObj)
{
  size_t nloc = dkmetaObj->nameloc.size();
  for (size_t iloc = 0; iloc < nloc; ++iloc ) {
    // skip calculation for random location ... should already be filled
    const std::string rkey = "random decay";
    if ( dkmetaObj->nameloc[iloc] == rkey ) {
      if ( iloc != 0 ) {
        std::cerr << "calcLocationWeights \"" << rkey << "\""
                  << " isn't the 0-th entry" << std::endl;
        assert(0);
      }
      if ( dk2nuObj->nuenergy.size() != 1 ) {
        std::cerr << "calcLocationWeights \"" << rkey << "\""
                  << " nuenergy[" << iloc << "] not filled" << std::endl;
        assert(0);
      }
      continue;
    }
    TVector3 xyzDet(dkmetaObj->xloc[iloc],
                    dkmetaObj->yloc[iloc],
                    dkmetaObj->zloc[iloc]);  // position to evaluate
    double enu_xy = 0;  // give a default value
    double wgt_xy = 0;  // give a default value
    int status = CalcEnuWgt(dk2nuObj,xyzDet,enu_xy,wgt_xy);
    if ( status != 0 ) {
      std::cerr << "CalcEnuWgt returned " << status << " for "
                << dkmetaObj->nameloc[iloc] << std::endl;
    }
    // with the recalculated energy compute the momentum components
    TVector3 xyzDk(dk2nuObj->vx,dk2nuObj->vy,dk2nuObj->vz);  // origin of decay
    TVector3 p3 = enu_xy * (xyzDet - xyzDk).Unit();
    dk2nuObj->nupx.push_back(p3.x());
    dk2nuObj->nupy.push_back(p3.y());
    dk2nuObj->nupz.push_back(p3.z());
    dk2nuObj->nuenergy.push_back(enu_xy);
    dk2nuObj->nuwgt.push_back(wgt_xy);
  }
}

//____________________________________________________________________________
int CalcEnuWgt(const dk2nu* dk2nuObj, const TVector3& xyz,
               double& enu, double& wgt_xy)
{
  // Neutrino Energy and Weight at arbitrary point
  // Based on:
  //   NuMI-NOTE-BEAM-0109 (MINOS DocDB # 109)
  //   Title:   Neutrino Beam Simulation using PAW with Weighted Monte Carlos
  //   Author:  Rick Milburn
  //   Date:    1995-10-01

  // History:
  // jzh  3/21/96 grab R.H.Milburn's weighing routine
  // jzh  5/ 9/96 substantially modify the weighting function use dot product
  //              instead of rotation vecs to get theta get all info except
  //              det from ADAMO banks neutrino parent is in Particle.inc
  //              Add weighting factor for polarized muon decay
  // jzh  4/17/97 convert more code to double precision because of problems
  //              with Enu>30 GeV
  // rwh 10/ 9/08 transliterate function from f77 to C++

  // Original function description:
  //   Real function for use with PAW Ntuple To transform from destination
  //   detector geometry to the unit sphere moving with decaying hadron with
  //   velocity v, BETA=v/c, etc..  For (pseudo)scalar hadrons the decays will
  //   be isotropic in this  sphere so the fractional area (out of 4-pi) is the
  //   fraction of decays that hit the target.  For a given target point and
  //   area, and given x-y components of decay transverse location and slope,
  //   and given decay distance from target ans given decay GAMMA and
  //   rest-frame neutrino energy, the lab energy at the target and the
  //   fractional solid angle in the rest-frame are determined.
  //   For muon decays, correction for non-isotropic nature of decay is done.

  // Arguments:
  //    dk2nu    :: contains current decay information
  //    xyz      :: 3-vector of position to evaluate
  //                in *beam* frame coordinates  (cm units)
  //    enu      :: resulting energy
  //    wgt_xy   :: resulting weight
  // Return:
  //    (int)    :: error code
  // Assumptions:
  //    Energies given in GeV
  //    Particle codes have been translated from GEANT into PDG codes

  // for now ... these masses _should_ come from TDatabasePDG
  // but use these hard-coded values to "exactly" reproduce old code
  //
  const double kPIMASS = 0.13957;
  const double kKMASS  = 0.49368;
  const double kK0MASS = 0.49767;
  const double kMUMASS = 0.105658389;
  const double kOMEGAMASS = 1.67245;

  const int kpdg_nue       =   12;  // extended Geant 53
  const int kpdg_nuebar    =  -12;  // extended Geant 52
  const int kpdg_numu      =   14;  // extended Geant 56
  const int kpdg_numubar   =  -14;  // extended Geant 55

  const int kpdg_muplus     =   -13;  // Geant  5
  const int kpdg_muminus    =    13;  // Geant  6
  const int kpdg_pionplus   =   211;  // Geant  8
  const int kpdg_pionminus  =  -211;  // Geant  9
  const int kpdg_k0long     =   130;  // Geant 10  ( K0=311, K0S=310 )
  const int kpdg_k0short    =   310;  // Geant 16
  const int kpdg_k0mix      =   311;
  const int kpdg_kaonplus   =   321;  // Geant 11
  const int kpdg_kaonminus  =  -321;  // Geant 12
```

```
  const int kpdg_omegaminus =  3334;  // Geant 24
  const int kpdg_omegaplus  = -3334;  // Geant 32

  const double kRDET = 100.0;   // set to flux per 100 cm radius

  double xpos = xyz.X();
  double ypos = xyz.Y();
  double zpos = xyz.Z();

  enu    = 0.0;  // don't know what the final value is
  wgt_xy = 0.0;  // but set these in case we return early due to error


  // in principle we should get these from the particle DB
  // but for consistency testing use the hardcoded values
  double parent_mass = kPIMASS;
  switch ( dk2nuObj->ptype ) {
  case kpdg_pionplus:
  case kpdg_pionminus:
    parent_mass = kPIMASS;
    break;
  case kpdg_kaonplus:
  case kpdg_kaonminus:
    parent_mass = kKMASS;
    break;
  case kpdg_k0long:
  case kpdg_k0short:
  case kpdg_k0mix:
    parent_mass = kK0MASS;
    break;
  case kpdg_muplus:
  case kpdg_muminus:
    parent_mass = kMUMASS;
    break;
  case kpdg_omegaminus:
  case kpdg_omegaplus:
    parent_mass = kOMEGAMASS;
    break;
  default:
    std::cerr << "CalcEnuWgt unknown particle type " << dk2nuObj->ptype
              << std::endl << std::flush;
    assert(0);
    return 1;
  }

  double parentp2 = ( dk2nuObj->pdpx*dk2nuObj->pdpx +
                      dk2nuObj->pdpy*dk2nuObj->pdpy +
                      dk2nuObj->pdpz*dk2nuObj->pdpz );
  double parent_energy = TMath::Sqrt( parentp2 +
                                     parent_mass*parent_mass);
  double parentp = TMath::Sqrt( parentp2 );

  double gamma     = parent_energy / parent_mass;
  double gamma_sqr = gamma * gamma;
  double beta_mag  = TMath::Sqrt( ( gamma_sqr - 1.0 )/gamma_sqr );

  // Get the neutrino energy in the parent decay CM
  double enuzr = dk2nuObj->necm;
  // Get angle from parent line of flight to chosen point in beam frame
  double rad = TMath::Sqrt( (xpos-dk2nuObj->vx)*(xpos-dk2nuObj->vx) +
                            (ypos-dk2nuObj->vy)*(ypos-dk2nuObj->vy) +
                            (zpos-dk2nuObj->vz)*(zpos-dk2nuObj->vz) );

  double emrat = 1.0;
  double costh_pardet = -999., theta_pardet = -999.;

  // boost correction, but only if parent hasn't stopped
  if ( parentp > 0. ) {
    costh_pardet = ( dk2nuObj->pdpx*(xpos-dk2nuObj->vx) +
                     dk2nuObj->pdpy*(ypos-dk2nuObj->vy) +
                     dk2nuObj->pdpz*(zpos-dk2nuObj->vz) )
                     / ( parentp * rad);
    if ( costh_pardet >  1.0 ) costh_pardet =  1.0;
    if ( costh_pardet < -1.0 ) costh_pardet = -1.0;
    theta_pardet = TMath::ACos(costh_pardet);

    // Weighted neutrino energy in beam, approx, good for small theta
    emrat = 1.0 / ( gamma * ( 1.0 - beta_mag * costh_pardet ));
  }

  enu = emrat * enuzr;  // the energy ... normally

  // Get solid angle/4pi for detector element
  double sangdet = ( kRDET*kRDET /
                     ( (zpos-dk2nuObj->vz)*(zpos-dk2nuObj->vz) ) ) / 4.0;

  // Weight for solid angle and lorentz boost
  wgt_xy = sangdet * ( emrat * emrat );  // ! the weight ... normally

  // Done for all except polarized muon decay
  // in which case need to modify weight
  // (must be done in double precision)
  if ( dk2nuObj->ptype == kpdg_muplus || dk2nuObj->ptype == kpdg_muminus) {
    double beta[3], p_dcm_nu[4], p_nu[3], p_pcm_mp[3], partial;

    // Boost neu neutrino to mu decay CM
    beta[0] = dk2nuObj->pdpx / parent_energy;
    beta[1] = dk2nuObj->pdpy / parent_energy;
    beta[2] = dk2nuObj->pdpz / parent_energy;
    p_nu[0] = (xpos-dk2nuObj->vx)*enu/rad;
    p_nu[1] = (ypos-dk2nuObj->vy)*enu/rad;
    p_nu[2] = (zpos-dk2nuObj->vz)*enu/rad;
    partial = gamma *
      (beta[0]*p_nu[0] + beta[1]*p_nu[1] + beta[2]*p_nu[2] );
    partial = enu - partial/(gamma+1.0);
    // the following calculation is numerically imprecise
    // especially p_dcm_nu[2] leads to taking the difference of numbers
    //  of order ~10's and getting results of order ~0.02's
    // for g3numi we're starting with floats (ie. good to ~1 part in 10^7)
    p_dcm_nu[0] = p_nu[0] - beta[0]*gamma*partial;
    p_dcm_nu[1] = p_nu[1] - beta[1]*gamma*partial;
    p_dcm_nu[2] = p_nu[2] - beta[2]*gamma*partial;
    p_dcm_nu[3] = TMath::Sqrt( p_dcm_nu[0]*p_dcm_nu[0] +
                               p_dcm_nu[1]*p_dcm_nu[1] +
                               p_dcm_nu[2]*p_dcm_nu[2] );

    // Boost parent of mu to mu production CM
    double particle_energy = dk2nuObj->ppenergy;
    gamma = particle_energy/parent_mass;
    beta[0] = dk2nuObj->ppdxdz * dk2nuObj->pppz / particle_energy;
    beta[1] = dk2nuObj->ppdydz * dk2nuObj->pppz / particle_energy;
    beta[2] =                    dk2nuObj->pppz / particle_energy;
    partial = gamma * ( beta[0]*dk2nuObj->muparpx +
                        beta[1]*dk2nuObj->muparpy +
                        beta[2]*dk2nuObj->muparpz );
    partial = dk2nuObj->mupare - partial/(gamma+1.0);
    p_pcm_mp[0] = dk2nuObj->muparpx - beta[0]*gamma*partial;
    p_pcm_mp[1] = dk2nuObj->muparpy - beta[1]*gamma*partial;
    p_pcm_mp[2] = dk2nuObj->muparpz - beta[2]*gamma*partial;
    double p_pcm = TMath::Sqrt ( p_pcm_mp[0]*p_pcm_mp[0] +
                                 p_pcm_mp[1]*p_pcm_mp[1] +
                                 p_pcm_mp[2]*p_pcm_mp[2] );

    const double eps = 1.0e-30;  // ? what value to use
    if ( p_pcm < eps || p_dcm_nu[3] < eps ) {
      return 3; // mu missing parent info?
    }
    // Calc new decay angle w.r.t. (anti)spin direction
    double costh = ( p_dcm_nu[0]*p_pcm_mp[0] +
                     p_dcm_nu[1]*p_pcm_mp[1] +
                     p_dcm_nu[2]*p_pcm_mp[2] ) /
                   ( p_dcm_nu[3]*p_pcm );
    if ( costh >  1.0 ) costh =  1.0;
    if ( costh < -1.0 ) costh = -1.0;
    // Calc relative weight due to angle difference
    double wgt_ratio = 0.0;
    switch ( dk2nuObj->ntype ) {
    case kpdg_nue:
    case kpdg_nuebar:
      wgt_ratio = 1.0 - costh;
      break;
    case kpdg_numu:
    case kpdg_numubar:
    {
      double xnu = 2.0 * enuzr / kMUMASS;
      wgt_ratio = ( (3.0-2.0*xnu )  - (1.0-2.0*xnu)*costh ) / (3.0-2.0*xnu);
      break;
    }
    default:
      return 2; // bad neutrino type
    }
    wgt_xy = wgt_xy * wgt_ratio;
```

```

  } // ptype is muon

  return 0;
}

```

4.4 example input location file: `locfile.txt`

```
location in beam coordinates (cm)   tag
  0.1234    0.567    100000.   MINOS NearDet
  0.9999    0.987654321   735.0e5    MINOS FarDet      "
100.42   20.31415     80000.   "bogus position that I made up'
200.84   20.12121     500.another bogus position
```

4.5 output when running test script

```
$ root -b -q test_fill_dk2nu.C+
root [0]
Processing test_fill_dk2nu.C+...
Read 5 locations read from "locfile.txt"
{        0,         0,         0 } "random decay"
{   0.1234,     0.567,    100000 } "MINOS NearDet"
{   0.9999,  0.987654,  7.35e+07 } "MINOS FarDet"
{   100.42,   20.3142,     80000 } "bogus position that I made up"
{   200.84,   20.1212,       500 } "another bogus position"
```

5 Example use of the tree in a ROOT session

```
TFile* myfile = TFile::Open("test_dk2nu.root","READONLY");
TTree* mytree = 0;
myfile->GetObject("dk2nu",mytree);
mytree->Scan("run:evtno:@apdg.size():apdg[2]");
```

The `@` in `@apdg.size()` is the ROOT mechanism for signaling that the `.size()` method is to be applied to the collection as a whole and not on individual items, so this prints the length of the `apdg` STL `vector`. The `apdg[2]` prints the 3rd entry (if it exists); using `[]` (or giving none) for vectors performs an implicit loop. The looping rules for `Scan()` or `Draw()` on array elements in `TTree`s are complex and appropriate documentation should be consulted[1].

[1]http://root.cern.ch/root/html/TTree.html#TTree:Draw@2

6 Auxillary numbering schemes

Ndecay	Process
1	$K_L^0 \to \nu_e + \pi^- + e^+$
2	$K_L^0 \to \bar{\nu}_e + \pi^+ + e^-$
3	$K_L^0 \to \nu_\mu + \pi^- + \mu^+$
4	$K_L^0 \to \bar{\nu}_\mu + \pi^+ + \mu^-$
5	$K^+ \to \nu_\mu + \mu^+$
6	$K^+ \to \nu_e + \pi^0 + e^+$
7	$K^+ \to \nu_\mu + \pi^0 + \mu^+$
8	$K^- \to \bar{\nu}_\mu + \mu^-$
9	$K^- \to \bar{\nu}_e + \pi^0 + e^-$
10	$K^- \to \bar{\nu}_\mu + \pi^0 + \mu^-$
11	$\mu^+ \to \bar{\nu}_\mu + \nu_e + e^+$
12	$\mu^- \to \nu + \bar{\nu}_e + e^-$
13	$\pi^+ \to \nu_\mu + \mu^+$
14	$\pi^- \to \bar{\nu}_\mu + \mu^-$
999	Other

Code	Material
5	Beryllium
6	Carbon
9	Aluminum
10	Iron
11	Slab Steel
12	Blu Steel
15	Air
16	Vacuum
17	Concrete
18	Target
19	Rebar Concrete
20	Shotcrete
21	Variable Density Aluminum
22	Variable Density Steel
23	1018 Steel
24	A500 Steel
25	Water
26	M1018 Steel
28	Decay Pipe Vacuum
31	CT852

Table 11: The decay codes stored in `ndecay` and material codes as defined by Gnumi and used in the fluxfiles, old and current.

A Current Methods of Defining the `TTree`

A.1 geant3 based `gnumi`

The `gnumi` (GEANT3) ntuple is created using `hbook` as a column-wise (`common block`-based) ntuple. The ROOT version is generated by using `h2root` to convert it from the ZEBRA file format. As generation of new beamline simulations using this code is unlikely we will not further comment on the necessary steps for converting to the new format (it would be difficult).

A.2 `flugg`

The `flugg` `TTree` is filled using the script `numisoft/g4numi_flugg/root/fill_flux.C` which reads data from an ASCII text file. The extra ("extended") elements discussed in Table 6 are calculated when creating the entry; they are also apparently partially *kaput* (it's a technical term) due to a cut-and-paste typo.

With a minor reworking of the code the script could be rewritten to use compiled code and the actual structure. The would be the preferred route forward. The framework for this upgrade can be found in Section 4.

```
...
  TFile *ft = new TFile(ftree,"recreate");
  TTree *mtree = new TTree("h10","neutrino");
  int    run;      mtree->Branch("run",      &run,      "run/I");     //1
  int    evtno;    mtree->Branch("evtno",    &evtno,    "evtno/I");   //2
...
  double Ndxdznea; mtree->Branch("Ndxdznea", &Ndxdznea, "Ndxdznea/D");//7
...
  int events = 0;
  while(!datafile.eof()) {
    // read a line from the text file
       datafile
           >> run      //1
           >> evtno    //2
           ...
           >> beampz ; //62
...
       mtree->Fill();
       ++events;
  }
  datafile.close();
  mtree->Write();
  ft->Close();
```

To make this work for the new file format using the current approach basically involve changing the branch names, adding new branches and changing the types for those that are fixed sized arrays, making them vectors. This is probably not the right approach as it has no real benefits. Untested code follows:

```
#include <string>
#include <vector>
using namespace std;
...
  int bufsiz = 32000;  // best value?
  int splitlvl = 99;   // best value?
...
  std::vector<double> ndxdznear;
  mtree->Branch("ndxdznear","vector<double>",  &ndxdznear, bufsiz, splitlvl);
  ndxdznear.reserve(1);  // we know there will always be only one value (flugg files)
                         // and we must reserve space to have somewhere to put the value
                         // (this is less intensive than clear/push_back pairs in the loop)
...
  while(!datafile.eof()) {
    // read a line from the text file
...
      >> ndxdznear[0]  // already reserved space, so we can set it
...
```

An inspection of this script (`numisoft/g4numi_flugg/root/fill_flux.C`) turned up an error that needs to be fixed and committed back to all repository instances. The error is an obvious cut-and-paste typo:

```
  if (extend) {
     Vr = SumSq(Vx, Vy);
     pdPt = SumSq(pdPx, pdPy);
     pdP = SumSq(pdPt, pdPz);
     pppt = SumSq(ppdxdz, ppdydz)*pppz;
     ppp = SumSq(pppt, pppz);
     ppvr = SumSq(ppvx, ppvy);
     muparpt = SumSq(muparpx, muparpy);
     muparp = SumSq(muparpt, muparpz);
     ppvr = SumSq(tvx, tvy);  // the left hand side of this assignment should be "tvr"
                              // and not a repeat of "ppvr"
     tpt = SumSq(tpx, tpy);
     tp = SumSq(tpt, tpz);
  }
```

A.3 g4numi and variants

The g4numi TTree is filled in compiled code in numisoft/g4numi/src/NumiAnalysis.cc. The basic TTree is simply the series of data_t class objects, and is booked and filled via:

```
NumiAnalysis::NumiAnalysis()
...
    // individual entries in the tree are "data_t" objects
    g4data = new data_t();  // this is a private data member

void NumiAnalysis::book()
...
    nuNtuple = new TFile(nuNtupleFileName,"RECREATE","root ntuple");
    tree = new TTree("nudata","g4numi Neutrino ntuple");
    tree->Branch("data","data_t",&g4data,32000,1);

void NumiAnalysis::FillNeutrinoNtuple(const G4Track& ...
...
     // set values in g4data
     g4data->run = ...
     ...// loop for elements that are arrays
       g4data->NdxdzNear[ii] = ...
     ...
     tree->Fill();

void NumiAnalysis::finish()
...
nuNtuple->cd();
tree->Write();
nuNtuple->Close();
delete nuNtuple;
```

This is essentially the approach used in the Section 4 example. A couple of issues, as currently implemented, with this approach that I've noticed include:

1. the version number in the data_t.hh has never been incremented even when the layout changes (i.e. ClassDef(data_t,1) in data_t.hh always). In this scheme one really needs to always be sure to increment the version number whenever the data layout changes.

2. g4data->Clear() is never called, which means that entries that that vary in length (i.e. most of the MINERνA additions) retain high water values beyond the current ntrajectory from previous entries. This isn't an issue if one never indexes into the array beyond the current entry's set of values, but it can be confusing and it will cause the file to be larger than necessary (random values don't compress as well as 0).

The new ntuple format would be simply replacing the data_t with a new class. Member variable names would need adjustments in the NumiAnalysis code. Additionally, one would want to apply the Clear() method before the fill, which should reset any STL vectors to have zero length. Any instances of using fixed indexing during filling would need to be converted to push_back() methods on the element, i.e.:

```
    //OLD: g4data->NdxdzNear[ii] = ...
    dk2nu->ndxdznear.push_back(...);
```